LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-213454

Fast sparse matrix-vector multiplication by exploiting variable block structure

R. W. Vuduc, H.-J. Moon

July 8, 2005

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Fast sparse matrix-vector multiplication by exploiting variable block structure

Richard W. Vuduc* Hyun Jin Moon†

July 8, 2005

Abstract

We improve the performance of sparse matrix-vector multiply (SpMV) on modern cache-based superscalar machines when the matrix structure consists of multiple, irregularly aligned rectangular blocks. Matrices from finite element modeling applications often have this kind of structure. Our technique splits the matrix, A, into a sum, $A_1 + A_2 + \ldots + A_s$, where each term is stored in a new data structure, *unaligned block compressed sparse row (UBCSR) format* . The classical alternative approach of storing A in a block compressed sparse row (BCSR) format yields limited performance gains because it imposes a particular alignment of the matrix non-zero structure, leading to extra work from explicitly padded zeros. Combining splitting and UBCSR reduces this extra work while retaining the generally lower memory bandwidth requirements and register-level tiling opportunities of BCSR. Using application test matrices, we show empirically that speedups can be as high as $2.1\times$ over not blocking at all, and as high as $1.8\times$ over the standard BCSR implementation used in prior work. When performance does not improve, split UBCSR can still significantly reduce matrix storage.

Through extensive experiments, we further show that the empirically optimal number of splittings s and the block size for each matrix term A_i will in practice depend on the matrix and hardware platform. Our data lay a foundation for future development of fully automated methods for tuning these parameters.

1 Introduction

Although sparse matrix-vector multiply (SpMV) dominates the performance of diverse applications in scientific computing, economic modeling, and information retrieval (among others), conventional implementations have historically run at 10% of peak or less on uniprocessors [35]. Achieving higher performance requires choosing a compact data structure and appropriate code transformations that best exploit properties of both the sparse matrix—which may be known only at run-time—*and* the underlying machine architecture. The conventional data structure, compressed sparse row (CSR) format, stores each non-zero value along with an integer index to denote its position in the matrix. Thus, sparse kernels incur more computational overhead per non-zero matrix entry than their dense counterparts—overheads in the form of extra instructions and, critically, extra indirect and

*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P.O. Box 808, L-365, Livermore, California 94550, USA. E-mail: `richie@llnl.gov`. (**Corresponding author**)

†Department of Computer Science, University of California, Los Angeles, Los Angeles, California 90024, USA. E-mail: `hjmoon@cs.ucla.edu`

irregular memory accesses. We and others have studied a wide variety of techniques both for selecting data structures to reduce this overhead and for automatically tuning the resulting implementations (Section 6). In the best cases, tuned implementations of SpMV achieve up to 31% of machine peak and speedups as high as $4\times$ over CSR [15, 35, 37].

The best performance occurs for finite element method (FEM) modeling applications, but within this class there is a performance gap between matrices whose assembled non-zero structure consists primarily of dense blocks of a single size, uniformly aligned, and matrices whose structure consists of multiple block sizes with irregular alignment. Figure 1 shows the structural differences: Figure 1 (*left*) shows a matrix whose non-zero pattern (blue dots) consists entirely of 8×8 dense blocks, uniformly aligned with respect to the rows and columns, while Figure 1 (*right*) has more complex dense substructure. Users typically exploit the structure of Figure 1 (*left*) using block compressed sparse row (BCSR) format, which stores the matrix as a sequence of fixed-size $r\times c$ dense blocks, with roughly one integer index stored per block instead of one per non-zero. Compared to CSR, BCSR reduces the number of indices by $\frac{1}{rc}$, and the fixed block size enables unrolling and register-level tiling of each block-multiply. However, two difficulties arise in practice:

1. **The best $r\times c$ varies both by matrix *and* by machine.** Contrary to what most users would reasonably expect, 8×8 is not the best block size on many platforms in the seemingly simple case of Figure 1 (*left*) [35]. This fact motivates automatic tuning.

2. **Any speedup is mitigated by explicit zero padding needed to obtain uniform dense block substructure.** On a Pentium III architecture, applying BCSR to Figure 1 (*right*) reduces the execution time of SpMV to $\frac{2}{3}$ that of CSR, for a $1.5\times$ speedup. However, this technique must store explicit zeros to fill the blocks, here incurring 50% more flops [35]. Can we better capture the structure and choose a data structure that reduces this extra work, yielding possibly even better speedups? This paper considers (a) *splitting* the sparse matrix A into the sum $A = A_1 + \cdots + A_s$, where each

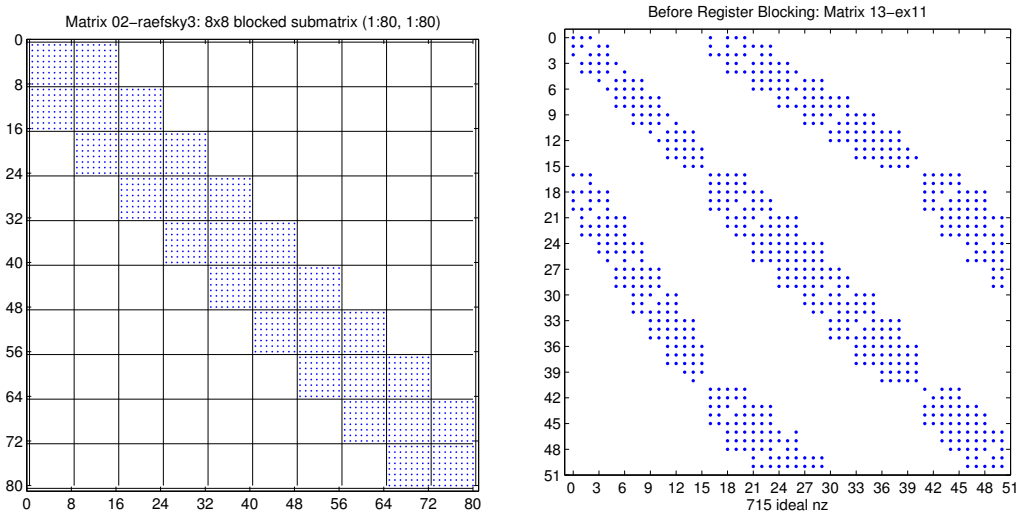

Figure 1: **Block structure in practice**. (Left) An 80×80 submatrix taken from a larger sparse matrix in a fluid flow simulation based on the finite element method. Each dot is one non-zero entry, and the non-zero pattern consists entirely of 8×8 dense blocks, uniformly aligned as shown. (Right) A 50×50 submatrix from a different FEM fluid flow simulation.

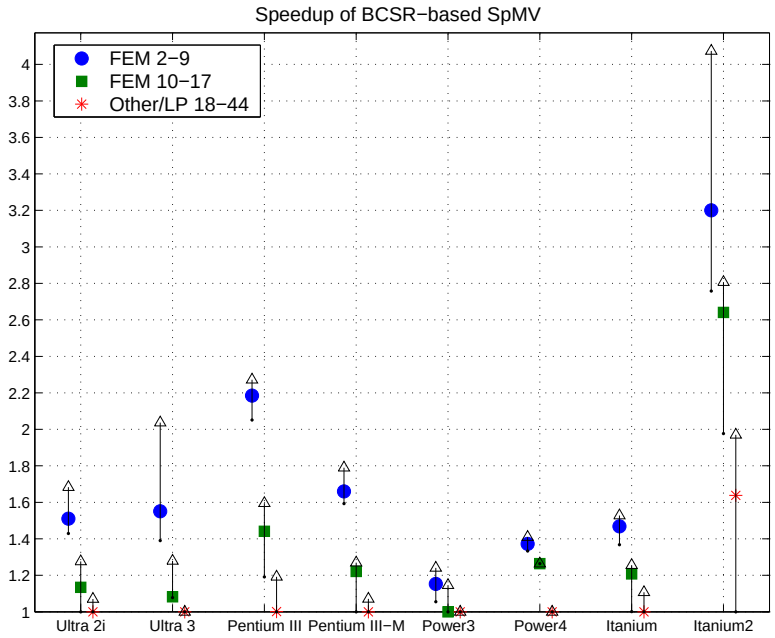

Figure 2: **The speedup gap among application classes across platforms**. We show minimum, maximum, and median summary statistics of BCSR speedup relative to CSR (*i.e.*, unblocked 1×1 BCSR). For each platform, we separate data by application. Median fractions for FEM Matrices 2–9 are shown by blue solid circles, for FEM Matrices 10–17 by green solid squares, and for all remaining matrices by red asterisks. Arrows indicate a range from the minimum fraction to the maximum.

> A_i may be stored with a different block size, and (b) store each A_i in a flexible *unaligned block compressed sparse row (UBCSR) format* that relaxes *both* row and column alignment conditions of BCSR, at the cost of indirect access to both x and y, instead of just x as in BCSR and CSR.

How large is the gap in performance between the two FEM classes in practice? Figure 2, taken from Vuduc [35, Chap. 4], summarizes experimentally observed speedups of BCSR-based SpMV compared to a conventional CSR implementation on 8 hardware platforms using a set of 44 test matrices comprising the SPARSITY benchmark suite [15]. We divide these matrices into three classes: FEM matrices 2–9, whose structure resembles Figure 1 (*left*), FEM matrices 10–17, whose structure resembles Figure 1 (*right*), and matrices from all other applications (*e.g.*, economic modeling, chemical process simulation, linear programming).[1] For each class on each platform, we show the minimum and maximum speedup as the end points of an arrow, and the median speedup within a class by a colored marker. The achieved speedups of the three classes are clearly separated, and the median speedups for FEM 2–9 are between $1.1\times$ and $1.54\times$ higher than the median speedups for FEM 10–17. This paper shows how split UBCSR can reduce this gap. An important by-product of the split formulation is a significant reduction in matrix storage.

Furthermore, we show our split UBCSR SpMV runs in practice in as little as half the time of CSR ($2.1\times$ speedup) and $\frac{5}{9}$ the time of BCSR ($1.8\times$ speedup) on a variety of application matrices with complex structure like that shown in Figure 1 (*right*) (Section 5). These

[1] Matrix 1 is a dense matrix stored in sparse format, and not a true application matrix. We therefore omit it.

results are empirical, collected through experiments based on exhaustive search of many possible implementations, each tunable for a given matrix and machine by the number of splitting terms s and the block size for each term. Our characterizations of application matrices and our data on split UBCSR performance gathered from these experiments, when combined with prior work on automatic tuning, should lead to fully automated heuristics for tuning these parameters. We are making our techniques available in the Optimized Sparse Kernel Interface (OSKI) [36], a library of automatically tuned sparse matrix kernels that builds on SPARSITY, an earlier prototype [15, 14].

2 Characterizing Variable Block Structure in Practice

Section 1 notes the performance gap between two subclasses of finite element method (FEM) matrices (Matrices 2–9 *vs.* 10–17). To distinguish the dense block substructure of these classes, we use variable block row (VBR) format [28, 26]. Roughly speaking, VBR defines the matrix block structure by logically partitioning rows into block rows and columns into block columns. In VBR, we find that Matrices 10–17 differ from 2–9 in two ways:[2]

- **Unaligned blocks**: Consider a dense $r \times c$ subblock whose upper-leftmost entry is (i, j), where the matrix is $m \times n$, $0 \leq i < m$, and $0 \leq j < n$. Then, block compressed sparse row (BCSR) format assumes a *uniform alignment constraint* in which all blocks have $i \bmod r = j \bmod c = 0$. Relaxing just the column alignment (by allowing $j \bmod c$ to be any value less than c) has in practice yielded modest improvements [7, 21]. When Matrices 12 and 13 are stored in VBR format, we find most non-zeros are contained in blocks of the same size, but $i \bmod r$ and $j \bmod c$ are distributed uniformly over all possible values up to $r - 1$ and $c - 1$, respectively. Our unaligned block compressed sparse row (UBCSR) format relaxes row alignments as well.

- **Mixtures of "natural" block sizes**: Matrices 10, 15, and 17 possess a mix of block sizes, at least when viewed in VBR format. This motivates decomposing the nonzero substructure such that $A = A_1 + A_2 + \cdots + A_s$, where each term A_i consists of the subset of blocks of a particular size. Each term can then be tuned separately.

We present these observations for the matrices listed in Table 1, which includes a subset of the matrices referred to previously as Matrices 10–17, as well as 5 additional matrices (labeled Matrices A–E) from other FEM applications.

2.1 The FEM performance gap

To see the FEM gap, consider the performance of Matrix 12 using BCSR compared to the best BCSR performance on FEM Matrices 2–9. Table 2 compares the best observed performance for Matrix 12 (column 3) to both a reference implementation using compressed sparse row (CSR) format storage (column 6) and the best observed performance on Matrices 2–9 (column 2) several platforms. Performance is measured in Mflop/s, but does *not* count flops due to fill (*i.e.*, the flops on explicitly filled in zeros); thus, the reported performance reflects inverse time. We show the best BCSR block size for Matrix 12 (column 4), and we show the corresponding *fill ratio*, or ratio of the number of stored entries including

[2] We treat Matrix 11, which contains a mix of blocks and diagonals, using different techniques described elsewhere [35, Chap. 5]; Matrices 14 and 16 are eliminated on our evaluation platforms due to their small size.

#	Matrix	Dimension	No. of Non-zeros	Dominant block sizes (% of non-zeros)
10	`ct20stif` Engine block	52329	2698463	6×6 (39%) 3×3 (15%)
12	`raefsky4` Buckling problem	19779	1328611	3×3 (96%)
13	`ex11` 3D flow	16614	1096948	1×1 (38%) 3×3 (23%)
15	`vavasis3` 2D partial differential equation	41092	1683902	2×1 (81%) 2×2 (19%)
17	`rim` Fluid mechanics problem	22560	1014951	1×1 (75%) 3×1 (12%)
A	`bmw7st_1` Car body analysis	141347	7339667	6×6 (82%)
B	`cop20k_m` Accelerator cavity design	121192	4826864	2×1 (26%), 1×2 (26%) 1×1 (26%), 2×2 (22%)
C	`pwtk` Pressurized wind tunnel	217918	11634424	6×6 (94%)
D	`rma10` Charleston Harbor model	46835	2374001	2×2 (17%) 3×2 (15%), 2×3 (15%) 4×2 (9%), 2×4 (9%)
E	`s3dkq4m2` Cylindrical shell	90449	4820891	6×6 (99%)

Table 1: **Variable block test matrices**. Matrices with variable block structure and/or non-uniform alignment. Dominant block sizes $r \times c$ are shown in the last column, along with the percentage of non-zeros contained within $r \times c$ blocks shown in parentheses. The sources of these problems is summarized elsewhere [35, Appendix B].

explicit zeros to the number of true non-zero entries, at those block szies (column 5). In all cases, we observe speedups over the reference. However, if we compute the fraction of the best performance on Matrices 2–9 (by dividing column 3 by column 2) and then take the median over all platforms, we find the median fraction to be only 69%.

In fact, Matrix 12 has a rich block structure which BCSR is not capturing precisely. Figure 3 shows the 51×51 submatrix beginning at the $(715, 715)$ entry of Matrix 12. In the left plot, we superimpose the logical grid of 2×2 cells that would be imposed in BCSR, and in the right plot we superimpose the grid of 3×3 cells. These block sizes require significant fill-in of explicit zeros: the ratios of stored entries to true non-zeros, or the *fill ratios*, are 1.46 for 3×3, and 1.24 for 2×2. The desire to reduce this extra storage and work motivates the techniques of this paper.

2.2 Characterizing block alignments

We can capture the block structure of Matrix 12 more precisely by storing the matrix in VBR and analyzing its structure. (See Appendix A for a detailed description of VBR.) Figure 4 (*top*) shows the same 51×51 submatrix shown in Figure 3 as it would be blocked in VBR. We used a routine from the SPARSKIT library to convert the matrix from CSR to

Platform	Matrices 2–9 Maximum Mflop/s	Matrix 12-raefsky4 Best Mflop/s	$r \times c$	Fill	1×1 Mflop/s
Ultra 2i	57	38	2×2	1.24	33
Ultra 3	109	61	2×1	1.13	56
Pentium III	90	63	3×3	1.46	40
Pentium III-M	120	83	2×2	1.24	68
Power3	168	130	1×1	1.00	130
Itanium 1	214	172	3×1	1.24	140
Itanium 2	1122	774	4×2	1.48	276

Table 2: **FEM performance gap when using BCSR format: Matrix 12-raefsky4**. We show the best performance using BCSR (column 3), the best register block size $r_{\text{opt}} \times c_{\text{opt}}$ (column 4), and the fill ratio at $r_{\text{opt}} \times c_{\text{opt}}$ (column 5) for Matrix 12-raefsky4. This example shows the typical gap between performance achieved on Matrices 10–17 and the best performance on Matrices 2–9 (column 1). This data is taken from Vuduc [35, Chap. 4].

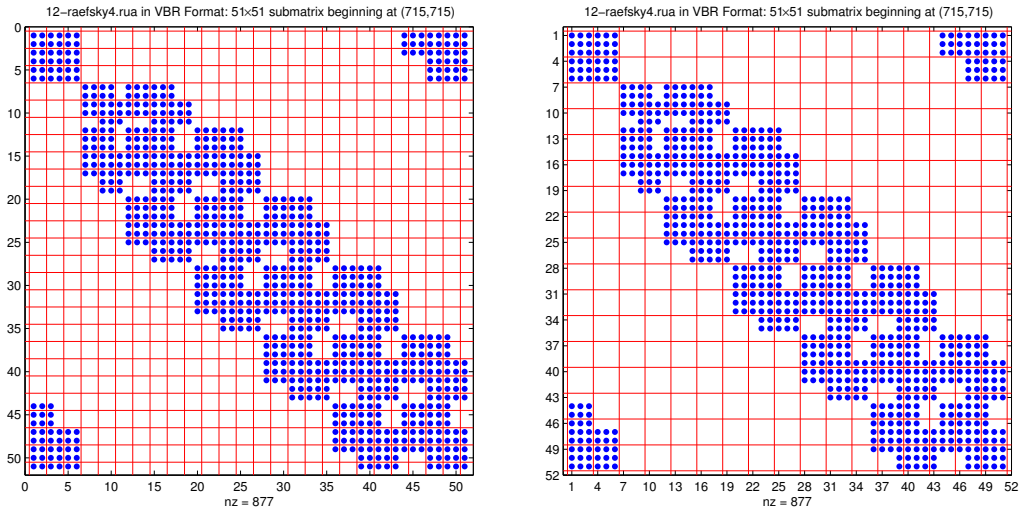

Figure 3: **Uniform block sizes can inadequately capture "natural" block structure**. We show the 51×51 submatrix beginning at element (715, 715) of Matrix 12-raefsky4 when uniformly aligned 2×2 (*left*) and 3×3 (*right*) logical grids have been imposed, as occurs when using BCSR. These grids do not precisely capture the true non-zero structure, leading to fill ratios of 1.23 for 2×2 blocking, and 1.46 for 3×3 blocking.

VBR format [28]. This routine partitions the rows by looping over rows in order, starting at the first row, and placing rows with identical non-zero structure in the same block. The same procedure is used to partition the columns. The distribution of non-zeros can be obtained in one pass over the resulting VBR data structure. For Matrix 12, the maximum block size in VBR format turns out to be 3×3. In Figure 4 (*bottom-left*), we show the fraction of non-zeros contained in all blocks of a given size $r\times c$, where $1 \leq r, c \leq 3$. Each square represents a value of $r\times c$ shaded by the fraction of non-zeros for which it accounts, and labeled by that fraction. A label of '0' indicates that the fraction is zero when rounded to two digits, but there is at least 1 block at the given size. For Matrix 12, 96% of the non-zeros occur in 3×3 blocks.

Although Matrix 12 is dominated by 3×3 blocks, these blocks are not uniformly aligned on row boundaries as assumed by BCSR. In Figure 4 (*bottom-right*), we show the distributions of $i \bmod r$ and $j \bmod c$, where (i, j) is the starting position in A of each 3×3 block, and the top-leftmost entry of A is $A(0, 0)$. The first row index of a given block row can start on any alignment, with 26% of block rows having $i \bmod r = 1$, and the remainder split equally between $i \bmod r = 0$ and 2. This observation motivates the use of a format like UBCSR which allows more flexible alignments.

We summarize the variable block structure of the matrix test set used in this paper in the rightmost column of Table 1. This table includes a short list of dominant block sizes after conversion to VBR format, along with the fraction of non-zeros for which those block sizes account. The reader may assume that the dominant block size is also irregularly aligned except in the case of Matrix 15. More information on the distribution of non-zeros and block size alignments appears elsewhere [35, Appendix F].

2.3 Characterizing mixed block structure

Some matrices, when stored in VBR, contain mixtures of block sizes. Figure 5 (*top*) shows the distribution of blocks for Matrix D in Table 1. The dominant block size is 2×2, but these blocks account for only 17% of the total non-zeros. Moreover, these block sizes may also have arbitrary alignments as shown for 2×2 blocks in this matrix in Figure 5 (*bottom*). Again, VBR provides one useful way to identify the existence of such structure.

3 A Split Unaligned Block Matrix Representation

For matrices with mixed or irregularly aligned dense block substructure, we consider the following implementation of sparse matrix-vector multiply (SpMV):

1. Convert or store the matrix A first in VBR format. We modify the default SPARSKIT CSR-to-VBR conversion procedure to allow some fill in of explicit zeros as a way of changing the block size distribution.

2. Split A into a sum of s terms, $A = A_1 + \cdots + A_s$, according to the distribution of block sizes observed when A is in VBR. We could theoretically select one term for each block size in the distribution, though in practice we may not want to do so.

3. Store each term A_i in UBCSR format, an unaligned version of the traditional BCSR format. UBCSR augments BCSR with an additional array of row indices, thereby allowing each block row to start at any index. Thus, we only constrain all blocks in a given block row to have a particular alignment.

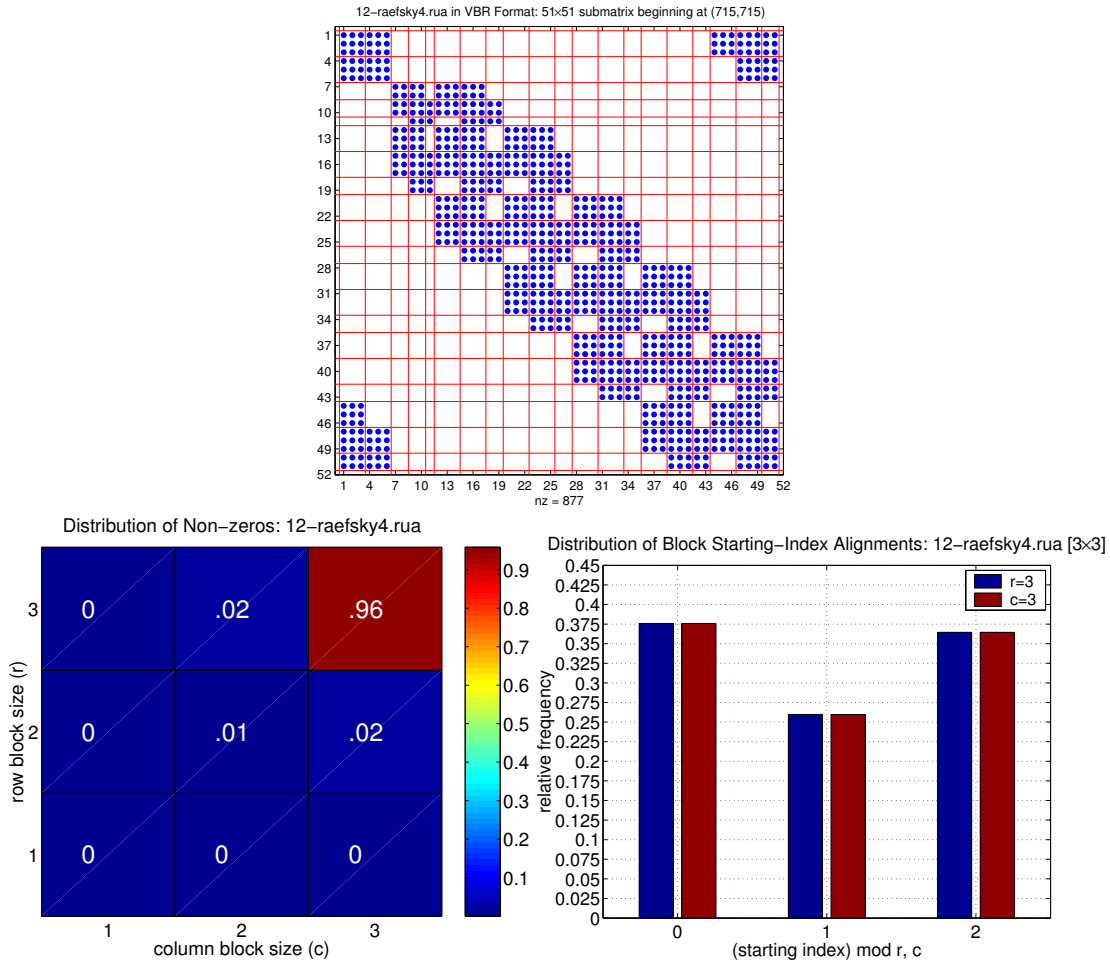

Figure 4: **Logical grid (block partitioning) after greedy conversion to variable block row (VBR) format: Matrix 12-`raefsky4`.** (*Top*) We show the logical block partitioning after conversion to VBR format using a greedy algorithm. (*Bottom-left*) Approximately 96% of the non-zero blocks are 3×3. (*Bottom-right*) Let (i, j) be the starting row and column index of each 3×3 block. We see that 37.5% of these blocks have $i \bmod 3 = 0$, 26% have $i \bmod 3 = 1$, and the remaining 36.5% have $i \bmod 3 = 2$. The starting column indices follow the same distribution, since the matrix is structurally (though not numerically) symmetric.

The use of VBR is a *heuristic* for identifying block structure. The problem of finding the maximum number of non-overlapping dense blocks in a matrix is NP-Complete [34], so there is considerable additional scope for analyzing dense block structure.

Although VBR is useful for characterizing the structure as done in Section 2, VBR implementations of SpMV typically have poor performance in practice. The innermost loops of the typical VBR implementation carry out multiplication by an $r\times c$ block. However, this block multiply cannot be unrolled in the same way as BCSR because the column block size c may change from block to block within a block row. Performance in this format on uniprocessors falls well below that of alternative formats [35, Chap. 2].

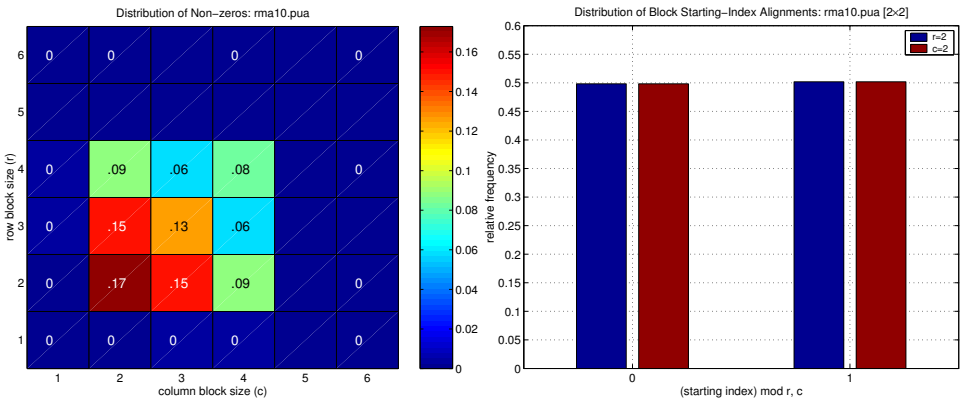

Figure 5: **Distribution and alignment of block sizes: Matrix `rma10`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 2×2 blocks. Specifically, we plot the fraction of 2×2 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

3.1 Converting to variable block row format with fill

The default SPARSKIT CSR-to-VBR conversion routine only groups rows (or columns) when the non-zero patterns between rows (columns) matches exactly. However, this convention can be too strict on some matrices in which it would be profitable to fill in zeros, just as with BCSR. Below, we discuss a simple variation on the SPARSKIT routine that allows us to create a partitioning based on a measure of similarity between rows (columns).

First, consider the example of Matrix 13. Table 1 indicates that this matrix has relatively few block sizes larger than the trivial unit block size (1×1). However, the 52×52 submatrix of Matrix 13, depicted in Figure 6 shows that a few isolated zero elements break up potentially larger blocks.

We relax the default partitioning using the following measure of similarity between columns (or equivalently, rows). Let u and v be two sparse column vectors whose non-zero elements are equal to 1. Let k_u and k_v be the number of non-zeros in u and v, respectively. Let $S(u, v)$ be the following measure of similarity between u and v:

$$S(u, v) = \frac{u^T \cdot v}{\max(k_u, k_v)} \tag{1}$$

This function is symmetric with respect to u and v, has a minimum value of 0 when u and v have no non-zeros in common, and a maximum value of 1 when u and v are identical.

The following procedure uses this similarity measure to compute a block row partitioning of an $m\times n$ sparse matrix A. We assume that A is a pattern matrix, *i.e.*, all non-zero entries are equal to 1. This partitioning is a set of lists of the rows of A, where rows in each list are in the same block row. On input, the caller provides a threshold, θ, specifying the minimum value of $S(u, v)$ at which two rows may be considered as belonging to the same block row. The procedure greedily examines rows sequentially, starting at row 0, and maintains a list of all row indices `Cur_block` in the current block row. Each row is

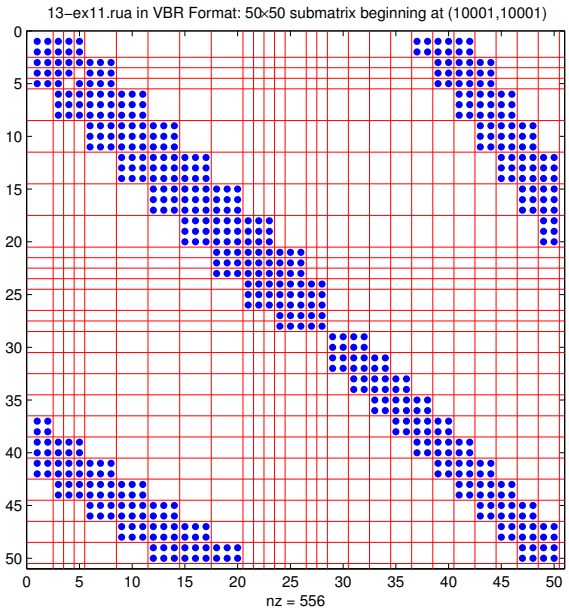

Figure 6: **Logical grid (block partitioning) after greedy conversion to VBR format: Matrix 13-ex11**. We show a 50×50 submatrix beginning at position (10001, 10001) in Matrix 13-ex11. The existence of explicit zero entries like those shown in the upper-left corner in positions (4,5) and (5,4) "break-up" the following contiguous blocks: one beginning at (39,3) and ending at (44,5), and another extending from (3,39) to (5,44).

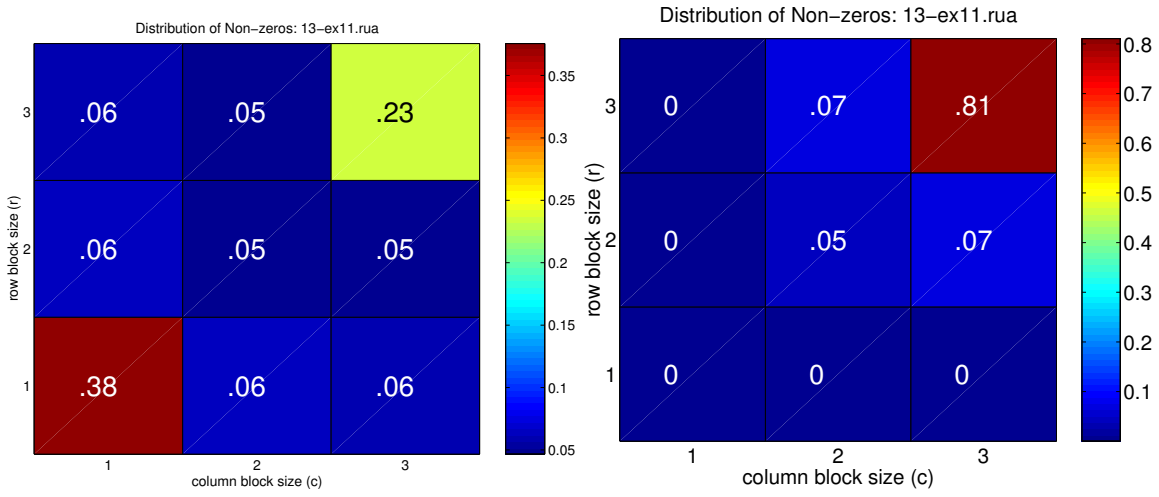

Figure 7: **Distribution of non-zeros over block sizes in variable block row format, without and with thresholding: Matrix 13-ex11**. (*Left*) Distribution of non-zeros without thresholding, *i.e.*, $\theta = 1$. Under the block partitioning shown in Figure 6, 23% of non-zeros are contained in 3×3 blocks, and 38% in 1×1 blocks. (*Right*) Distribution with thresholding at $\theta = 0.7$. The fraction of non-zeros in 3×3 blocks increases to 81%. The fill ratio is 1.01.

compared to the first row of the current block, and if their similarity exceeds θ, the row is added to the current block row. Otherwise, the procedure starts a new block row.

```
    Procedure PartitionRows( A, θ )
1       Cur_block ← [0] /* Ordered list of row indices in current block */
2       All_blocks ← ∅
3       for i = 1 to m − 1 do /* Loop over rows */
4           Let u ← row Cur_block[0] of A /* First row in current block */
5           Let v ← row i of A
6           if S(u^T, v^T) ≥ θ then
7               Append i onto Cur_block
            else
8               All_blocks ← All_blocks ∪ Cur_block
9               Cur_block ← [i]
10      All_blocks ← All_blocks ∪ Cur_block
11      return All_blocks
```

We may partition the columns using a similar procedure. However, all of the matrices in Table 1 are structurally (but not numerically) symmetric, so the row partition can be used as a column partition. The SPARSKIT CSR-to-VBR routine can take these row and column partitions as inputs, and returns A in VBR format. The conversion routine fills in explicit zeros to make the blocks conform to the partitions.

Applying Procedure **PartitionRows** to Matrix 13 with $\theta = 0.7$ shifts the non-zero distribution so that 3×3 blocks contain 81% of all stored values (including filled in zeros), instead of just 23% when $\theta = 1$. The fill ratio (stored values including filled in zeros divided by true number of non-zeros) is 1.01 at $\theta = 0.7$. Thus, more opportunities for blocking become available at the cost of only a 1% increase in flops. Section 4 discusses how we chose θ in our experiments.

3.2 Splitting the non-zero pattern

We handle multiple blocks sizes by first computing the distribution of work (*i.e.*, non-zero elements) over block sizes from the VBR data structure, and then computing an s-way splitting of the matrix A into a sum of matrices $A = A_1 + \ldots + A_s$, where A_i holds all non-zeros of a particular block size and is stored in UBCSR format. We consider structurally disjoint splittings, *i.e.*, A_i and A_j have no non-zeros in common when $i \neq j$.

Given s, a list of block sizes for each term A_i, and a fill threshold θ (Section 3.1), we consider each term A_i in turn, repeatedly extracting all blocks of size $r_i \times c_i$ and returning the remaining non-zeros in A_0:

```
    Procedure Split(A, θ, r_1, c_1, ..., r_s, c_s)
1       Let A_1, A_0 ← SplitOnce(A_0, θ, r_i, c_i)
2       for i = 2 to s − 1 do
3           A_i, A_0 ← SplitOnce(A_0, 1.0, r_i, c_i)
4       A_s ← A_0
5       return A_1, ..., A_s
```

This procedure uses θ only at the first splitting, although in principle one could use a different threshold at each i.

For each term A_i, the auxiliary procedure **SplitOnce** converts the current non-zero pattern A_0 to VBR using **PartitionRows** (Section 3.1), greedily extracts $r_i \times c_i$ blocks, stores these blocks in $\hat{A}$ in UBCSR format, and finally returns $\hat{A}$ and the leftover non-zeros:

```
    Procedure SplitOnce (A, θ, r, c)
1       Let V ← A converted to VBR format at threshold θ
2       Let Â ← empty matrix
3       foreach block row I in V, in increasing order of row index do
4           foreach block b in I of size at least r×c,
                    in increasing order of column index do
5               Convert block b into as many non-overlapping but adjacent
                    r×c blocks as possible, with the first block aligned at the
                    upper left corner of b
6               Add these blocks to Â
7       return Â in UBCSR format, A − Â in CSR format
```

This procedure does not extract *exact* $r \times c$ blocks, but rather extracts as many non-overlapping $r \times c$ blocks as possible from any block of size at least $r \times c$ (lines 4–5).

Since **SplitOnce** is a greedy algorithm, the order in which the block sizes are specified matters. We specify how we deal with this issue experimentally in Section 4.

3.3 An unaligned block compressed sparse row format

We handle unaligned block rows in UBCSR format by simply augmenting the usual BCSR data structure with an additional array of row indices `Arowind` such that `Arowind`$[I]$ contains the starting index of block row I. Listing 1 shows an example of the 2×3 UBCSR code to compute the SpMV operation, $y \leftarrow y + A \cdot x$, where A is the sparse matrix and x, y are dense vectors. The array `Aval` stores the non-zero values block-by-block, where blocks from the same block row are stored consecutively and each block is stored in row-major order. The array `Acolind` stores the starting column index for each block. The array `Aptr` stores the start of each block row in `Aind`. Readers familiar with BCSR will recognize that Listing 1 differs from its BCSR counterpart essentially at lines 7 and 8, where we must load the elements of y indirectly for UBCSR.

4 Experimental Methods

The split UBCSR implementation of SpMV has the following parameters: the similarity threshold θ which controls fill, the number of splitting terms s, and the block sizes for all terms, $r_1 \times c_1, \ldots, r_s \times c_s$. Given a matrix and machine, we select these parameters by the empirical search procedure described in this section.

Such an exhaustive search is generally not practical at run-time, owing to the cost of conversion. For instance, the time to execute **SplitOnce** just once is roughly comparable in cost to the conversion cost observed for BCSR conversion—between 5–40 reference SpMVs [35]. While automated methods exist for selecting a block size in the BCSR case [7, 15, 35, 37], none exist for the split UBCSR case as far as we know. Thus, our results (Section 5) should be interpreted as an empirical upper-bound on how well we expect to be able to do. We believe the exhaustive experimental data will inform the future development of parameter selection heuristics in the split UBCSR case.

Listing 1: **Example 2x3 matrix-vector multiply implementation for sparse UBCSR matrices.** Multiplication by each block is fully unrolled (lines 16–18). Only the indirect load of y at lines 7–8 would differ in a BCSR implementation.

```
1  void sparse_mvm_ubcsr_2x3( int M, const double* Aval,
       const int* Arowind, const int* Acolind, const int* Aptr,
3      const double* x, double* y )
   {
5      int I;
       for( I = 0; I < M; I++ ) { /* loop over block rows */
7          double* yp = y + Arowind[I]; /* block row start */
           register double y0 = yp[0], y1 = yp[1];
9          int jj ;
           for( jj = Aptr[I]; jj < Aptr[I+1]; jj ++, Aval += 6 ) { /* loop over non-zero blocks */
11             const double* xp = x + Acolind[jj ];
               register double x0 = xp[0], x1 = xp[1], x2 = xp[2];
13             /* fully unrolled and register-tiled block multiply */
               y0 += Aval[0]*x0; y1 += Aval[3]*x0;
15             y0 += Aval[1]*x1; y1 += Aval[4]*x1;
               y0 += Aval[2]*x2; y1 += Aval[5]*x2;
17         } /* jj */
           yp[0] = y0;  yp[1] = y1;
19     } /* I */
   }
```

4.1 Choosing the similarity threshold, θ

We consider just two values of θ: $\theta = 1$ ("exact match" partitioning) and $\theta = \theta_{\min}$, chosen as follows. For all $\theta \in \Theta = \{0.5, 0.55, 0.6, \ldots, 1.0\}$, we compute the non-zero distribution over block sizes after conversion to VBR format. Denote the block sizes by $r_1 \times c_1, \ldots, r_t \times c_t$. Consider a splitting $A = A_1 + A_2 + \ldots + A_t$ at θ, where each term A_i contains only the non-zeros contained in block sizes that are exactly $r_i \times c_i$, stored in UBCSR format. Let $\theta_{\min} \in \Theta$ be the threshold that *minimizes* the total size (bytes) of the data structure needed to store all A_i under this splitting, including all indices. Only for Matrices 10, 13, and 17 is $\theta_{\min} \neq 1$, as indicated in Appendix B. We refer the reader elsewhere for the actual distributions at various values of θ [35, Appendix F].

4.2 Choosing the number of splittings, s

We consider $2 \leq s \leq 4$, where A_s always contains the "leftover" non-zeros stored in CSR format. We allow A_s to have no elements if a particular splitting leads to no 1×1 blocks; multiplication by A_s becomes a no-op in this case. Although it is possible to consider larger upper limits than 4, there were relatively few instances in which even $s = 4$ led to an empirically optimal implementation in our test suite (see Section 5).

4.3 Choosing the block sizes, $\{r_i \times c_i\}$

Given θ and s, we perform an exhaustive empirical search over possible block sizes for each term using the following procedure.

1. We convert A to VBR format once to obtain a distribution of non-zeros over block sizes as in Figure 5 (*left*), rank these block sizes in decreasing order of matrix non-zeros each contains, and consider the top 3.

   For Figure 5 (*left*), the top 3 block sizes are 2×2, 3×2, and 2×3.

2. We then compute a set B of candidate block sizes to use in the splitting based on the *factors* of the top 3 block sizes in the distribution.

   For example, suppose the top 3 block sizes are 2×2, 3×3, and 8×1. Then the set of all factors dividing the row block sizes are $R = \{1, 2, 3, 4, 8\}$, and the column factors are $C = \{1, 2, 3\}$. The set of candidate block sizes for each term is the cross-product, $B = R \times C - \{(1, 1)\}$, where we have removed the 1×1 block size because A_s will always be in CSR format.

3. For every subset $b = \{(r_1, c_1), \ldots, (r_{s-1}, c_{s-1})\} \subseteq B$, we measure the performance of split UBCSR using the block sizes b, again with A_s stored in CSR. There are $\binom{|B|}{s-1}$ such subsets.

   However, the result of **Split** depends on the *order* of the block sizes. Thus, we actually try not only all subsets of B, but all permutations of each $b \subseteq B$ as well.

5 Results and Discussion

We show that the split UBCSR implementation of Section 3 often improves performance relative to a traditional register-tiled BCSR implementation for the matrices in Table 1 and hardware platforms listed in Table 3. Even when performance does not improve significantly, we reduce the overall storage.

5.1 Experimental setup

The 10 test matrices are listed in Table 1, and the 4 test platforms in Table 3. All performance is reported in Mflop/s, but we *do not* count flops on explicitly filled non-zeros as work. Thus, for a given matrix, these Mflop/s may be regarded as inverse time. Each measurement is the median of 25 trials; the median values were always close to the minimums.

For a given matrix and platform, we are particularly interested in comparing the best split UBCSR implementation (see Section 4) against the best BCSR implementation. The best BCSR implementation is chosen by exhaustively searching all block sizes up to 12×12, as done in prior work [15, 35, 37]. We also sometimes refer to the BCSR implementation as the *register blocking* implementation following the convention of earlier work.

5.2 Analysis and discussion

The left plots of Figures 8–11 compare the performance of the following implementations, for each platform and matrix. (We omit matrices that fit within a machine's largest cache.)

- *Best BCSR implementation for a dense matrix* (black hollow square): We store a dense matrix in BCSR format, and report the best performance over all block sizes up to 12×12. This data point is a kind of empirical upper-bound on SpMV performance, since the matrix has no sparsity and no run-time irregular memory access.

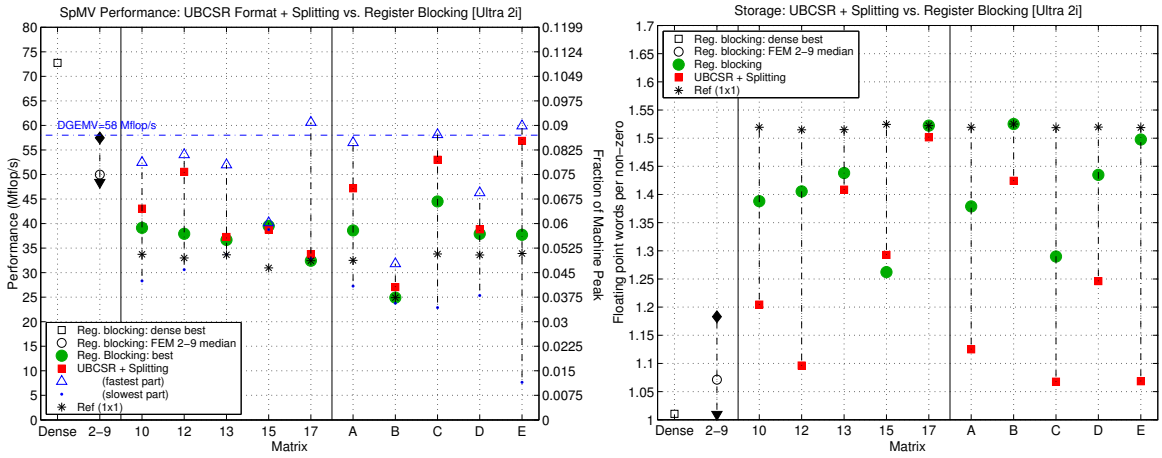

Figure 8: **Performance and storage for variable block matrices: Ultra 2i**. (*left*) Performance, Mflop/s. (*right*) Storage, in doubles per ideal non-zero. For additional data on the block sizes used, see Table 4, Appendix B.

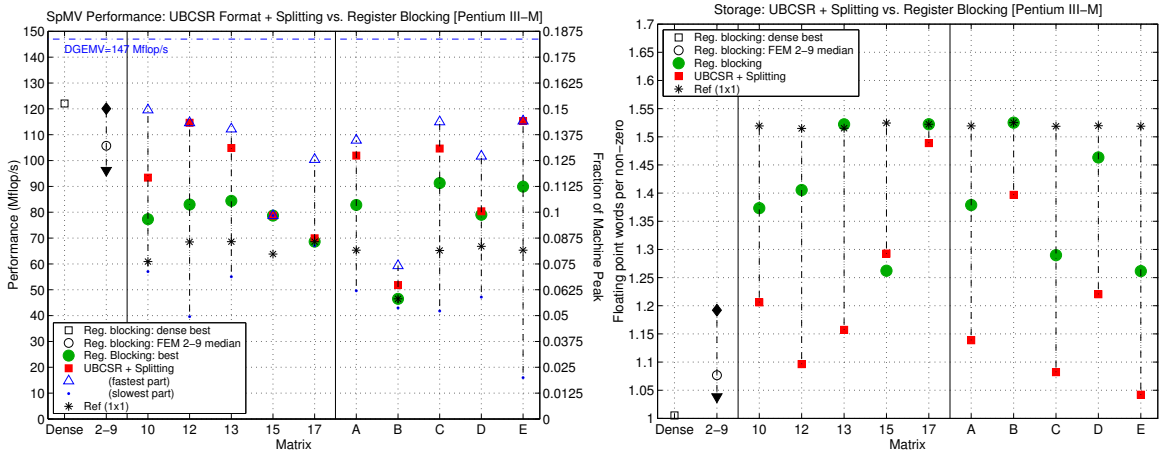

Figure 9: **Performance and storage for variable block matrices: Pentium III-M**. (*left*) Performance, Mflop/s. (*right*) Storage, in doubles per ideal non-zero. For additional data on the block sizes used, see Table 5, Appendix B.

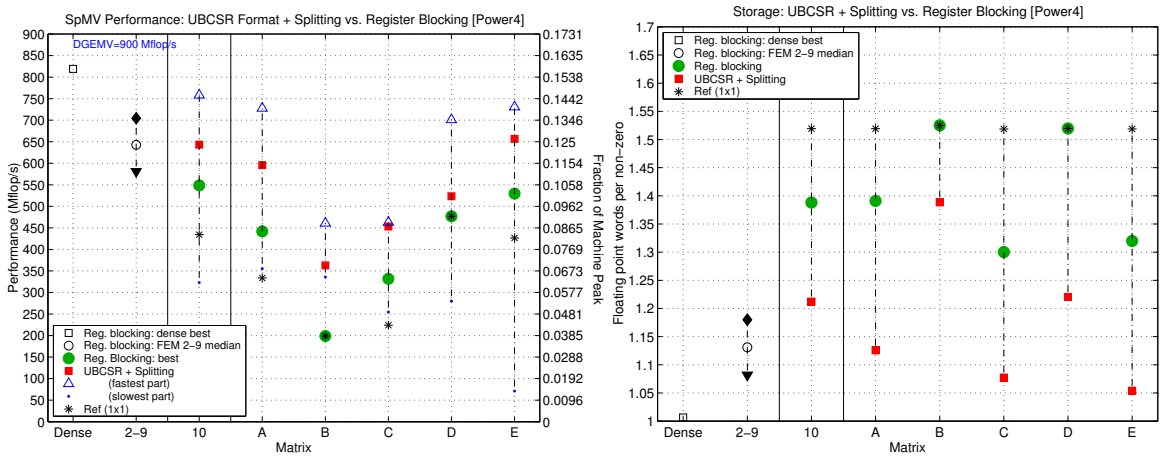

Figure 10: **Performance and storage for variable block matrices: Power4**. (*left*) Performance, Mflop/s. (*right*) Storage, in doubles per ideal non-zero. For additional data on the block sizes used, see Table 6, Appendix B.

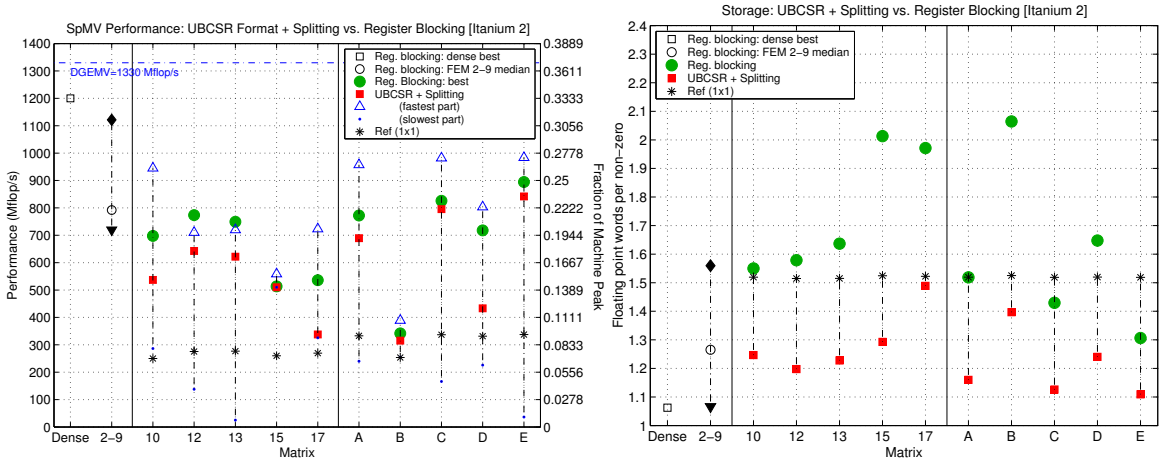

Figure 11: **Performance and storage for variable block matrices: Itanium 2**. (*left*) Performance, Mflop/s. (*right*) Storage, in doubles per ideal non-zero. For additional data on the block sizes used, see Table 7, Appendix B.

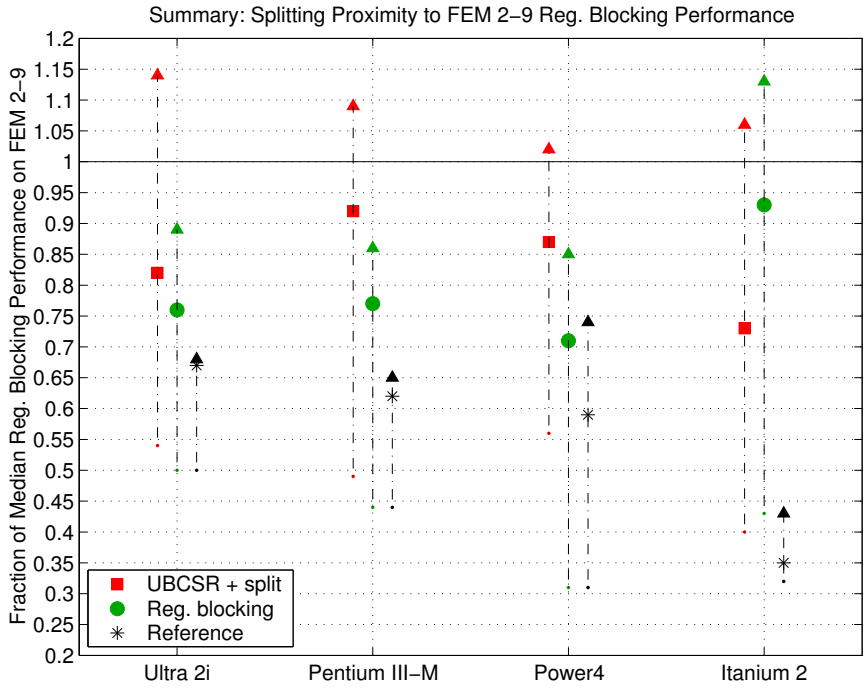

Figure 12: **Fraction of median BCSR performance over Matrices 2–9**.

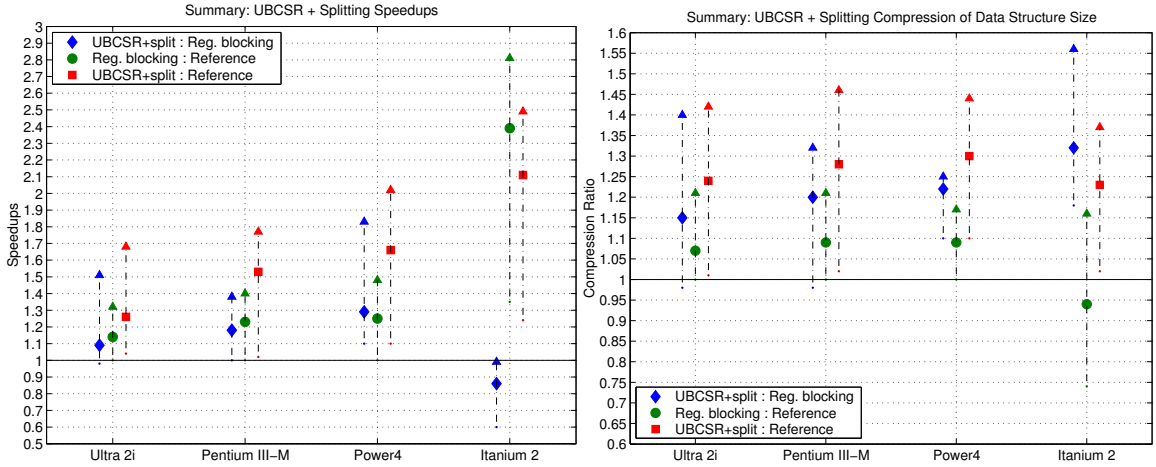

Figure 13: **Speedups and compression ratios after splitting + UBCSR storage, compared to BCSR**. (*Left*) We compare the following speedups: UBCSR storage + splitting over BCSR, BCSR over the reference CSR implementation, and UBCSR storage + splitting over the reference. For each platform, we show minimum, median, and maximum speedups for each pair. (*Right*) We compare the compression ratios for the same three pairs of implementations.

	Sun Ultra 2i	Intel Pentium III-M	IBM Power4	Intel Itanium 2
MHz	333	800	1300	900
OS	Solaris v5.8	Linux 2.4	AIX 5.2	Linux 2.4
Compiler	Sun cc v5.6	Intel C v8.1	IBM xlc v6	Intel C v7.1
Peak Mflop/s	667	800	5200	3600
DGEMM Mflop/s	425 ATLAS	640 Goto's	3500 ESSL	3500 Goto's
DGEMV Mflop/s	58 ATLAS	147 Intel MKL v5.2	900 ESSL	1330 Goto's
Peak MB/s	664	915	11000	6400
STREAM Triad MB/s	215	570	2286	4028
No. FP regs (double)	16	8	32	128
L1 size Line size Associativity	16 KB 16 B direct	16 KB 32 B 4-way	32 KB 128 B 2-way	32 KB 64 B 4-way
L2 size Line size Associativity	2 MB 64 B 2-way	256 KB 32 B 4-way	1.5 MB 128 B 8-way	256 KB 128 B 8-way
L3 size Line size Associativity	–	–	16 MB 512 B 8-way	1.5 MB 128 B 8-way
TLB entries Page size	64 8 KB	64 4 KB	1024 4 KB	128 16 KB
Memory size	256 MB	256 MB	4 GB	2 GB

Table 3: **Evaluation platforms**. In addition to machine parameters, we show dense BLAS performance for double-precision matrix-matrix multiply (DGEMM) and matrix-vector multiply (DGEMV), and sustainable bandwidth according to the STREAM benchmark.

- *Median, minimum, and maximum BCSR performance on Matrices 2–9* (black hollow circle, black solid diamond, and black solid downward-pointing triangle, respectively): For Matrices 2–9, consider the best BCSR performance observed after an exhaustive search. We show the median, minimum, and maximum of these values.
- *Splitting and UBCSR storage* (red solid squares): Performance of the best split UBCSR SpMV implementation, chosen by the limited search procedure in Section 4.
- *Fastest and slowest component under splitting* (blue triangle and dot): We measure the raw Mflop/s of SpMV for *each component* A_i. We show the fastest component by a blue triangle, the slowest by a blue dot, and the two components are connected by a vertical dash-dot line. These points suggest (indirectly) to what extent the fastest and slowest component of each splitting contributes to overall performance.
- *BCSR implementation* (green dots): Best BCSR performance from earlier reports [15, 35].
- *Reference* (black asterisks): Performance in CSR format.

For reference, we also show in the left subplots of Figures 8–11 the performance of tuned dense matrix-vector multiply (DGEMV) on a large (out-of-cache) matrix, shown by a blue horizontal dash-dot line (also labeled by performance in Mflop/s).

In the right subplots of Figures 8–11, we show the total size (in doubles) of the data structure normalized by the number of true non-zeros (*i.e.*, excluding fill).

We summarize the main observations as follows:

1. **By relaxing the block row alignment using UBCSR storage, it is possible to approach the performance seen on Matrices 2–9.** A single block size and irregular alignment characterize the structure of Matrices 12, 13, A, C, and E. The best absolute performance under splitting within a given platform is typically seen on these matrices. Furthermore, this performance is roughly comparable to median BCSR performance taken over Matrices 2–9 on the same platform. These two observations suggest that the overhead of the additional row indices in UBCSR is small. We summarize how closely the split implementations approach the performance observed for Matrices 2–9 in Figure 12, discussed in more detail below.

2. **Median speedups, taken over the matrices in Table 1 and measured relative to the reference performance, range from** $1.26\times$ **(Ultra 2i) up to** $2.1\times$ **(Itanium 2).** Furthermore, splitting can be up to $1.8\times$ faster than BCSR alone. We summarize the minimum, median, and maximum speedups in Figure 13 (*left*).

3. **Splitting can lead to a significant reduction in total matrix storage.** The compression ratio of splitting over the reference is the size of the reference (CSR) data structure divided by the size of the split+UBCSR data structure. The median compression ratios of splitting over the reference, taken over the matrices in Table 1, are between 1.26–$1.3\times$. Compared to BCSR, the compression ratios of splitting can be as high as $1.56\times$. We summarize the minimum, median, and maximum compression ratios in Figure 13 (*right*).

These three findings confirm the potential improvements in speed and storage using UBCSR format and splitting. We elaborate on these conclusions below.

5.2.1 Proximity to uniform BCSR performance

The performance under splitting and UBCSR storage can approach or even slightly exceed the median BCSR performance on FEM Matrices 2–9. For each platform, we show in Figure 12 the minimum, median, and maximum performance on the matrices in Table 1. Performance is displayed as a fraction of median BCSR performance taken over Matrices 2–9. We also show statistics for BCSR only and reference implementations. The median fraction achieved by splitting exceeds the median fraction achieved by BCSR on all but the Itanium 2. On the Pentium III-M and Power4, the median fraction of splitting exceeds the maximum of BCSR only, demonstrating the potential utility of splitting and the UBCSR format. The maximum fraction due to splitting slightly exceeds 1 on all platforms.

The data for the individual platforms, Figures 8–11 (*left*), shows that the best performance is attained on Matrices 12, 13, A, C, and E, which are all dominated by a single unaligned block size (see Table 1). However, the fastest component of the splitting is comparable in performance to the median FEM 2–9 performance in at least half the cases on all platforms. Not surprisingly, splitting performance can be limited by the slowest component, which in most cases is the CSR implementation, or in the case of Matrix 15, "small"

block sizes like 2×1 and 2×2. On Itanium 2, the fastest component is close to or in excess of the BCSR performance (Figure 11 (*left*)) but overall performance never exceeds BCSR performance. This observation suggests the importance of targeting the CSR (1×1) implementation for low-level tuning [35, Chap. 4].

5.2.2 Median speedups

We compare the following speedups on each platform in Figure 13 (*left*):

- Speedup of splitting over BCSR (blue solid diamonds)
- Speedup of BCSR over the reference (green solid circles)
- Speedup of splitting over the reference (red solid squares)

Figure 13 (*left*) shows minimum, median, and maximum speedups taken over the matrices in Table 1.

Splitting is at least as fast as BCSR on all but the the Itanium 2 platform. Median speedups, taken over the matrices in Table 1 and measured relative to the reference performance, range from $1.26\times$ (Ultra 2i) up to $2.1\times$ (Itanium 2). Relative to BCSR, median speedups are relatively modest, ranging from 1.1–$1.3\times$. However, these speedups can be as much as $1.8\times$ faster.

5.2.3 Reduced storage requirements

Though the speedups can be relatively modest, splitting can significantly reduce storage requirements. The asymptotic storage for CSR, ignoring row pointers, is 1.5 doubles per non-zero when the number of integers per double is 2 [35, Chap. 3]. When abundant dense blocks exist, the storage decreases toward a lower limit of 1 double per non-zero. Figures 8–11 (*right*) compare the storage per non-zero between CSR, BCSR, and the splitting implementations. We also show the minimum, median, and maximum storage per non-zero taken over FEM Matrices 2–9 for BCSR. Except for Matrix 15, splitting reduces the storage on all matrices and platforms, and is often comparable to the median storage requirement for Matrices 2–9.

In the case of Matrix 15, the slight increase in storage is due to a small overhead in UBCSR storage. All natural dense blocks are 2×1 or 2×2 and uniformly aligned for this matrix [35, Appendix F].

On Itanium 2, the dramatic speedups over the reference from BCSR come at the price of increased storage—just over 2 doubles per non-zero on Matrices 15, 17, and B. Though the splitting implementations are slower, they dramatically reduce the storage requirement in these cases.

We summarize the overall compression ratios across platforms in Figure 13 (*right*). We define the compression ratio for format a over format b as the size of the matrix in format b divided by the size in format a (larger ratios mean a requires less storage). We compare the compression ratio for the following pairs of formats in Figure 13 (*right*):

- Compression ratio of splitting over BCSR (blue solid diamonds)
- Compression ratio of BCSR over the reference (green solid circles)
- Compression ratio of splitting over the reference (red solid squares)

Median compression ratios, taken over the matrices in Table 1, for the split/UBCSR representation over BCSR range from 1.15 to 1.3. Relative to the reference, the median compression ratio for splitting ranges from 1.24 to 1.3, but can be as high as 1.45, which is close to the asymptotic limit.

6 Related Work

The inspiration for this study comes from recent work on splitting by Geus and Röllin [11], Pinar and Heath [25], and Toledo [33], and the performance gap we have observed informally [15, 37, 14]. Geus and Röllin explore up to 3-way splittings for a particular application matrix used in accelerator cavity design, but the splitting terms are still based on row-aligned BCSR format. (The last splitting term in their implementations is also fixed to be CSR, as in our work.) Pinar and Heath restrict their attention to 2-way splittings where the first term is $1\times c$ format and the second in 1×1. Toledo also considered 2-way splittings and block sizes up to 2×2, as well as low-level tuning techniques (*e.g.*, prefetching) to improve memory bandwidth. The main distinction of our work is relaxed row-alignment.

We use conversion to VBR as a heuristic for identifying existing block structure. Vassilevska and Pinar recently showed that finding the maximum number of non-overlapping dense blocks is NP-Complete [34], motivating the VBR-based analysis. Non-zero structure analysis tools developed by Bik and Wijshoff [4] and Knijnenburg and Wijshoff [18] provide a complementary means by which to detect and extract non-zero patterns.

Split UBCSR can be combined with other techniques that improve register-level reuse and reuse of the matrix entries, including multiplication by multiple vectors [14, 2]. Speedups of up to $7\times$ over CSR, and $2.5\times$ over the single-vector case have been observed [15, 19].

Section 2 notes the matrices were all structurally symmetric, and sometimes numerically symmetric as well. Symmetric BCSR storage yields speedups as high as $2.8\times$ over non-symmetric CSR, $2.1\times$ over non-symmetric BCSR, and of course significantly reduces storage [19]. Symmetric multiple vector implementations can be even faster [19].

While much of the work above focuses on exploiting blocks, diagonals and bands are another common substructure. The classical setting in which diagonal structure-centric formats like diagonal (DIAG) format and jagged diagonal (JAD) format have been applied is on vector architectures [38, 23, 24]. Recent work has shown the potential pay-offs from careful application on superscalar cache-based microprocessors [35, Chap. 5].

Pinar and Heath reorder rows and columns of a sparse matrix to *create* dense rectangular block structure which might then be exploited by splitting [25]. Their formulation is based on the Traveling Salesman Problem. In the context of SPARSITY, Moon, *et al.*, have applied this idea to the SPARSITY benchmark suite, showing speedups over conventional register blocking of up to $1.5\times$ on Matrices 17, 20, 21, and 40. Heras, *et al.*, have also proposed TSP-based reordering schemes, with an emphasis on theoretical aspects of formulating the problem [13]. Open issues include when to apply TSP-based reordering, what approximation heuristics are likely to work best, and what the run-time costs will be.

Related to these reordering techniques are classical methods that reduce the matrix bandwidth or fill for numerical factorization [8, 17, 1, 27, 5, 10, 31, 32]. Orderings have been applied to SpMV as well [16, 33, 6, 12]. Temam and Jalby have proven in a simple 1-level cache model that reducing bandwidth helps to minimize self-interference misses, suggesting additional careful study may be fruitful [30].

The class of matrices represented by Matrices 18–44 of the SPARSITY benchmark suite

largely remain difficult. One effective technique is cache-level blocking [22, 14]. Cache blocking reorganizes a large sparse matrix into a collection of smaller, disjoint rectangular blocks to improve temporal access to elements of x. The largest improvements occur on large, randomly structured matrices like linear programming Matrices 41–44 of the SPARSITY test suite, as well as matrices from latent semantic indexing applications [3].

Temam and Jalby propose an interesting and as-yet unexplored variation on cache blocking we refer to as *diagonal cache blocking* [30]. They show by a theoretical analysis in a simple cache model that reducing the bandwidth helps to minimize self-interferences misses. They further observe that blocking the matrix in "bands" achieves the same effect, though we are not aware of any empirical validation to date.

Better low-level tuning of the CSR SpMV implementation may also be possible. Recent work on low-level tuning of SpMV by unroll-and-jam (Mellor-Crummey, *et al.* [20]), software pipelining (Geus and Röllin [11]), and prefetching (Toledo [33]) are promising starting points. On the vector Cray X1, just one additional permutation of rows in CSR, with no other data reorganization, yields order of magnitude improvements [9].

Another promising line of research is considering how to reuse the matrix itself in the context of higher-level kernels and solvers, since just reading the matrix dominates the execution time of SpMV [37]. Examples include recent work on automatically tuning $A^TA \cdot x$ and powers of a matrix, $A^k \cdot x$ [29, 35].

7 Conclusions and Future Work

This paper extends the classical BCSR format to handle matrices with irregularly aligned and mixed dense block substructure, thereby reducing the gap between various classes of FEM matrices. We are making this new split UBCSR data structure available in the Optimized Sparse Kernel Interface (OSKI), a library of automatically tuned sparse matrix kernels that builds on SPARSITY, an earlier prototype [36, 15].

However, our results are really only empirical bounds on what may be possible since they are based on exhaustive search over split UBCSR's tuning parameters (Section 4). We are pursuing effective and cheap heuristics for selecting these parameters, and our data already suggest the form this heuristic might take. One key component is a cheap estimator of the non-zero distributions over block sizes, which we measured in this paper exactly using VBR. This estimator would be similar to those proposed in prior work for estimating fill in the BCSR case [15, 35], and would suggest the number of splittings and candidate block sizes. Earlier heuristic models for the BCSR case, which use benchmarking data to characterize the machine-specific performance at each block size, could be extended to the UBCSR case and combined with the estimation data [7, 15, 35].

We are also pursuing combining split UBCSR with many of the other SpMV optimizations surveyed in Section 6, including symmetry, multiple vectors, and cache blocking. In the case of cache blocking, CSR is often used as an auxiliary data structure; replacing the use of CSR with the 1×1 UBCSR data structure itself could reduce some of the row pointer overhead when the matrix is very sparse.

References

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[2] A. H. Baker, E. R. Jessup, and T. Manteuffel. A technique for accelerating the convergence of restarted GMRES. Technical Report CU-CS-045-03, University of Colorado, Dept. of Computer Science, January 2003.

[3] M. W. Berry, S. T. Dumais, and G. W. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37(4):573–595, 1995.

[4] A. J. C. Bik and H. A. G. Wijshoff. Automatic nonzero structure analysis. *SIAM Journal on Computing*, 28(5):1576–1587, 1999.

[5] I. Brainman and S. Toledo. Nested-dissection orderings for sparse LU with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 23(4):998–1012, May 2002.

[6] D. A. Burgess and M. B. Giles. Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines. Technical report, Numerical Analysis Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, 1995.

[7] A. Buttari, V. Eijkhout, J. Langou, and S. Filippone. Performance optimization and modeling of blocked sparse kernels. Technical Report ICL-UT-04-05, Innovative Computing Laboratory, University of Tennessee, Knoxville, 2005.

[8] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the ACM National Conference*, 1969.

[9] E. D'Azevedo, M. R. Fahey, and R. T. Mills. Vectorized sparse matrix multiply for compressed sparse row storage. In *Proceedings of the International Conference on Computational Science*, LNCS 3514, pages 99–106, Atlanta, GA, USA, May 2005. Springer.

[10] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, April 1973.

[11] R. Geus and S. Röllin. Towards a fast parallel sparse matrix-vector multiplication. In E. H. D'Hollander, J. R. Joubert, F. J. Peters, and H. Sips, editors, *Proceedings of the International Conference on Parallel Computing (ParCo)*, pages 308–315. Imperial College Press, 1999.

[12] G. Heber, R. Biswas, and G. R. Rao. Self-avoiding walks over adaptive unstructured grids. *Concurrency: Practice and Experience*, 12(2–3):85–109, 2000.

[13] D. B. Heras, V. B. Perez, J. C. C. Dominguez, and F. F. Rivera. Modeling and improving locality for irregular problems: sparse matrix-vector product on cache memories as a case study. In *HPCN Europe*, pages 201–210, 1999.

[14] E.-J. Im. *Optimizing the performance of sparse matrix-vector multiplication*. PhD thesis, University of California, Berkeley, May 2000.

[15] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, 2004.

[16] E.-J. Im and K. A. Yelick. Optimizing sparse matrix vector multiplication on SMPs. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, USA, March 1999.

[17] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, Minneapolis, MN, USA, 1995.

[18] P. Knijnenburg and H. A. G. Wijshoff. On improving data locality in sparse matrix computations. Technical Report 94-15, Dept. of Computer Science, Leiden University, Leiden, The Netherlands, 1994.

[19] B. C. Lee, R. Vuduc, J. Demmel, and K. Yelick. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *Proceedings of the International Conference on Parallel Processing*, Montreal, Canada, August 2004.

[20] J. Mellor-Crummey and J. Garvin. Optimizing sparse matrix vector multiply using unroll-and-jam. In *Proceedings of the Los Alamos Computer Science Institute Third Annual Symposium*, Santa Fe, NM, USA, October 2002.

[21] H. J. Moon, R. Vuduc, J. W. Demmel, and K. A. Yelick. Matrix splitting and reordering for sparse matrix-vector multiply. Technical Report (*to appear*), University of California, Berkeley, Berkeley, CA, USA, 2004.

[22] R. Nishtala, R. Vuduc, J. Demmel, and K. Yelick. When cache blocking sparse matrix vector multiply works and why. In *Proceedings of the PARA'04 Workshop on the State-of-the-art in Scientific Computing*, Copenhagen, Denmark, June 2004.

[23] H. Okuda, K. Nakajima, M. Iizuka, L. Chen, and H. Nakamura. Parallel finite element analysis platform for the Earth Simulator: GeoFEM. In P. M. A. Sloot, D. Abramson, A. V. Bogdanov, J. J. Dongarra, A. Y. Zomaya, and Y. E. Gorbachev, editors, *Proceedings of the 3rd International Conference on Computational Science Workshop on Computational Earthquake Physics and Solid Earth System Simulation*, volume III of *LNCS 2659*, pages 773–780, Melbourne, Australia, June 2003. Springer.

[24] H. Okuda, K. Nakajima, M. Iizuka, L. Chen, and H. Nakamura. Parallel finite element analysis platform for the Earth Simulator: GeoFEM. Technical Report 03-001, University of Tokyo, January 2003.

[25] A. Pinar and M. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of Supercomputing*, 1999.

[26] K. Remington and R. Pozo. NIST Sparse BLAS: User's Guide. Technical report, NIST, 1996. `gams.nist.gov/spblas`.

[27] D. J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. *Graph Theory and Computing*, pages 183–217, 1973.

[28] Y. Saad. SPARSKIT: A basic toolkit for sparse matrix computations, 1994. `www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html`.

[29] M. M. Strout, L. Carter, and J. Ferrante. Rescheduling for locality in sparse matrix computations. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 137–146, San Francisco, CA, May 2001. Springer.

[30] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of Supercomputing '92*, 1992.

[31] S.-H. Teng. Fast nested dissection on finite element meshes. *SIAM Journal on Matrix Analysis and Applications*, 18(3):552–565, 1997.

[32] W. F. Tinney and J. W. Walker. Direct solution of sparse network equations by optimally ordered triangular factorization. In *Proceedings of IEEE*, volume 55, pages 1801–1809, November 1967.

[33] S. Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[34] V. Vassilevska and A. Pinar. Finding nonoverlapping dense blocks of a sparse matrix. Technical Report LBNL-54498, Lawrence Berkeley National Laboratory, Berkeley, CA, USA, 2004.

[35] R. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, December 2003.

[36] R. Vuduc, J. Demmel, and K. Yelick. OSKI: An interface for a self-optimizing library of sparse matrix kernels, 2005. `bebop.cs.berkeley.edu/oski`.

[37] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.

[38] J. B. White and P. Sadayappan. On improving the performance of sparse matrix-vector multiplication. In *Proceedings of the International Conference on High-Performance C omputing*, 1997.

A Variable block row format

We illustrate VBR format in Figure 14, where we show a $m \times n = 6 \times 8$ matrix A containing $k = 19$ non-zeros. Consider a partitioning of this matrix into $M = 3$ block rows and $N = 4$ block columns as shown, yielding $K = 6$ blocks, each shaded with a different color. The VBR data structure is composed of the following 6 arrays:

- `brow` (length $M+1$): starting row positions in A of each block row. The I^{th} block row starts at row `brow`$[I]$ of A, ends at `brow`$[I+1]-1$, and `brow`$[M] = m$.
- `bcol` (length $N+1$): starting column positions in A of each block column. The J^{th} block column starts at column `bcol`$[J]$ of A, ends at `bcol`$[J+1]-1$, and `brow`$[N] = n$.
- `val` (length k): non-zero values, stored block-by-block. Blocks are laid out by row.

- `val_ptr` (length $K+1$): starting offsets of each block within `val`. The b^{th} block starts at position `val_ptr`$[b]$ in the array `val`. The last element `val_ptr`$[K] = k$.
- `ind` (length K): block column indices. The b^{th} block begins at column `bcol`$[$`ind`$[b]]$.
- `ptr` (length $M+1$): starting offsets of each block row within `ind`. The I^{th} block row starts at position `ptr`$[I]$ in `ind`.

The pseudo-code for SpMV using VBR appears below. Unlike BCSR, r and c are not fixed, making it difficult to unroll and register-tile line 11. We would also need to introduce branches to handle different fixed block sizes. The implementation in SPARSKIT uses 2-nested loops to perform the block multiply [28]. VBR performs poorly due to the overheads incurred by these loops.

```
      type brow : int[M + 1]
      type bcol : int[N + 1]
      type val : real[k]
      type val_ptr : int[K + 1]
      type ind : int[K]
      type ptr : int[M + 1]
1     foreach block row I do
2         i0 ← brow[I] /* starting row index */
3         r ← brow[I + 1] − brow[I] /* row block size */
4         Let ŷ ← y_{i0:(i0+r−1)}
5         for b = ptr[I] to ptr[I + 1] − 1 do /* blocks within I^th block row */
6             J ← ind[b] /* block column index */
7             j0 ← bcol[J] /* starting column index */
8             c ← bcol[J + 1] − bcol[J] /* column block size */
9             Let x̂ ← x_{j0:(j0+c−1)}
10            Let Â ← a_{i0:(i0+r−1),j0:(j0+c−1)}
                  /* Â = block of A stored in val[val_ptr[b] : (val_ptr[b + 1] − 1)] */
11            Perform r×c block multiply, ŷ ← ŷ + Â · x̂
12        Store ŷ
```

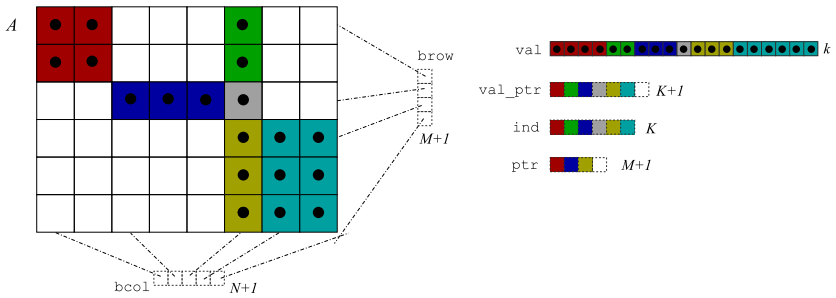

Figure 14: **Variable block row (VBR) format**. Here, a 6×9 sparse matrix with $k = 19$ non-zeros is partitioned into 3 block rows and 4 block columns, yielding 6 blocks.

	Register Blocking			Splitting					1×1
	Mflop/s	$r_{opt}\times c_{opt}$	Fill	Mflop/s	θ	$r_k\times c_k$	(stored nz) / (ideal nz)	Mflop/s	Mflop/s
10	39	2×1	1.10	43	0.9	3×3	0.77	52	34
						1×1	0.24	28	
				41	1	3×3	0.63	52	
						1×1	0.37	30	
12	38	2×2	1.24	51	1	3×3	0.96	54	33
						1×1	0.04	31	
13	37	2×1	1.14	37	1	3×1	0.34	52	34
						1×1	0.66	34	
15	40	2×1	1.00	39	1	2×1	1.00	40	31
						1×1	0.00	0	
17	32	1×1	1.00	34	0.8	2×1	0.24	61	32
						1×1	0.77	33	
				33	1	3×1	0.12	139	
						1×1	0.88	33	
A	39	2×2	1.22	47	1	6×6	0.82	57	32
						1×1	0.18	27	
B	25	1×1	1.00	27	1	2×1	0.48	32	25
						1×1	0.52	24	
C	44	3×3	1.22	53	1	6×6	0.94	58	34
						1×1	0.06	23	
D	38	2×1	1.14	39	1	2×2	0.77	46	34
						1×1	0.23	25	
E	38	8×2	1.45	57	1	6×2	0.99	60	34
						1×1	0.01	8	

Table 4: **Best unaligned block compressed sparse row splittings on variable block matrices, compared to register blocking: Ultra 2i**. Splitting data for Figure 8.

B Variable Block Splitting Data

Tables 4–7 show the splittings used in Figures 8–11. For each matrix (column 1), we show the following. Columns 2–4 show the best register blocking performance and corresponding block size, fill ratio. Columns 5–9 show the best performance with splitting, using UBCSR. The matrix is initially converted to VBR using a fill threshold of θ (column 6). We show the block size $r_k \times c_k$ used for each component of the splitting (column 7). We also show the corresponding number of non-zeros (divided by ideal non-zeros) for the k-th component (column 8), and estimated performance of just the k-th component (column 9) using the non-zero count in column 8. If the best splitting performance occurs for $\theta < 1$, we also show the data corresponding to the best performance when $\theta = 1$. Finally, column 10 shows the CSR reference performance.

	Register Blocking			Splitting					1×1
	Mflop/s	$r_{\text{opt}}\times c_{\text{opt}}$	Fill	Mflop/s	θ	$r_k\times c_k$	(stored nz) / (ideal nz)	Mflop/s	Mflop/s
10	77	2×2	1.21	93	0.9	3×3	0.77	120	61
						1×1	0.25	57	
				90	1	3×3	0.63	118	
						3×1	0.11	78	
						1×1	0.26	59	
12	83	2×2	1.24	115	1	3×3	0.96	111	68
						1×1	0.04	40	
13	84	3×2	1.40	105	0.7	3×3	0.82	112	69
						3×2	0.07	93	
						1×1	0.12	55	
				76	1	3×3	0.23	121	
						2×1	0.23	78	
						1×1	0.53	65	
15	79	2×1	1.00	79	1	2×1	1.00	79	64
						1×1	0.00	0	
17	69	1×1	1.00	70	0.8	4×1	0.17	100	69
						1×1	0.84	67	
				68	1	3×1	0.12	84	
						1×1	0.88	67	
A	83	2×2	1.22	102	1	3×3	0.88	108	65
						1×1	0.12	50	
B	46	1×1	1.00	52	1	2×1	0.48	59	46
						1×2	0.26	51	
						1×1	0.26	43	
C	91	3×3	1.22	105	1	3×6	0.94	115	65
						1×1	0.06	42	
D	79	2×2	1.29	80	1	3×2	0.45	102	67
						2×2	0.36	91	
						1×1	0.19	47	
E	90	2×2	1.11	115	1	6×6	0.99	114	65
						1×1	0.01	16	

Table 5: **Best unaligned block compressed sparse row splittings on variable block matrices, compared to register blocking: Pentium III-M**. Splitting data for Figure 9.

	Register Blocking			Splitting					1×1
	Mflop/s	$r_{\text{opt}}\times c_{\text{opt}}$	Fill	Mflop/s	θ	$r_k\times c_k$	(stored nz) / (ideal nz)	Mflop/s	Mflop/s
10	549	2×1	1.10	643	0.9	6×1	0.56	758	434
						3×1	0.30	681	
						2×1	0.09	623	
						1×1	0.07	323	
				579	1	3×2	0.61	703	
						3×1	0.13	590	
						1×1	0.26	414	
A	442	2×1	1.10	596	1	6×3	0.84	728	334
						1×1	0.16	355	
B	199	1×1	1.00	363	1	2×2	0.22	355	199
						2×1	0.26	461	
						1×2	0.26	354	
						1×1	0.26	336	
C	332	3×1	1.11	453	1	6×3	0.94	463	224
						1×1	0.06	254	
D	477	1×1	1.00	524	1	3×2	0.45	701	477
						2×2	0.36	550	
						1×1	0.19	280	
E	530	4×1	1.17	657	1	6×3	0.99	731	427
						1×1	0.01	71	

Table 6: **Best unaligned block compressed sparse row splittings on variable block matrices, compared to register blocking: Power4**. Splitting data for Figure 10.

	Register Blocking			Splitting					1×1
	Mflop/s	$r_{\text{opt}}\times c_{\text{opt}}$	Fill	Mflop/s	θ	$r_k\times c_k$	(stored nz) / (ideal nz)	Mflop/s	Mflop/s
10	698	4×2	1.45	537	1	6×1	0.49	945	250
						3×1	0.25	607	
						1×1	0.26	287	
12	774	4×2	1.48	643	1	3×1	0.97	710	276
						1×1	0.03	138	
13	749	4×2	1.54	622	0.7	3×1	0.89	720	277
						2×1	0.12	534	
						1×1	0.00	25	
				421	1	3×1	0.34	681	
						2×1	0.16	517	
						1×1	0.50	329	
15	514	4×1	1.78	510	1	2×1	1.00	559	260
						1×1	0.00	0	
17	536	4×1	1.75	338	0.8	4×1	0.17	723	269
						1×1	0.84	326	
				330	1	2×1	0.16	442	
						1×1	0.84	325	
A	772	4×2	1.43	689	1	6×1	0.87	957	333
						1×1	0.13	240	
B	342	2×2	1.82	315	1	2×1	0.48	389	254
						1×2	0.26	302	
						1×1	0.26	255	
C	826	4×2	1.34	795	1	6×1	0.95	982	337
						1×1	0.05	166	
D	718	4×2	1.55	433	1	4×2	0.21	804	331
						2×2	0.56	582	
						1×1	0.23	226	
E	895	4×2	1.23	842	1	6×1	0.99	984	337
						1×1	0.01	36	

Table 7: **Best unaligned block compressed sparse row splittings on variable block matrices, compared to register blocking: Itanium 2**. Splitting data for Figure 11.